Locating and Understanding Features of Complex Software Systems by Synchronizing Time-, Collaboration- and Code-focused Views on Execution Traces

Johannes Bohnet, Stefan Voigt, Jürgen Döllner
University of Potsdam
Hasso-Plattner-Institute
Prof.-Dr.-Helmert-Str. 2-3
14482 Potsdam, Germany
{bohnet, voigt, doellner}@hpi.uni-potsdam.de

Abstract

Extending or modifying features of complex software systems is often a highly time-consuming and cost-intensive task as, beforehand, the features have to be located within the code and to be understood in detail. To support developers in performing this task, we propose a technique that takes execution traces and implementation unit structuring as input data and provides various views thereupon. Views focus on different trace characteristics, namely they are time-, collaboration-, and code-focused. Synchronizing the views creates a rich user interface that helps developers to effectively identify and understand feature relevant parts of the implementation.

1. Introduction

How well programmers comprehend programs is key to effective software maintenance and evolution [11]. However, for complex software systems gathering insights into the system's structure and behavior is a highly time-consuming and cost-intensive task – even if only a small fraction of the system needs to be understood such as for the maintenance task of extending or modifying existing system features[1].

Implementing a feature change request means that developers need to understand (1) which design artifacts implement the feature and (2) how these artifacts collaborate to produce feature functionality. These tasks of *feature location* and *feature comprehension* are often difficult with complex software systems due to a lack of reliable high-level descriptions documenting traceability links from features to design artifacts.

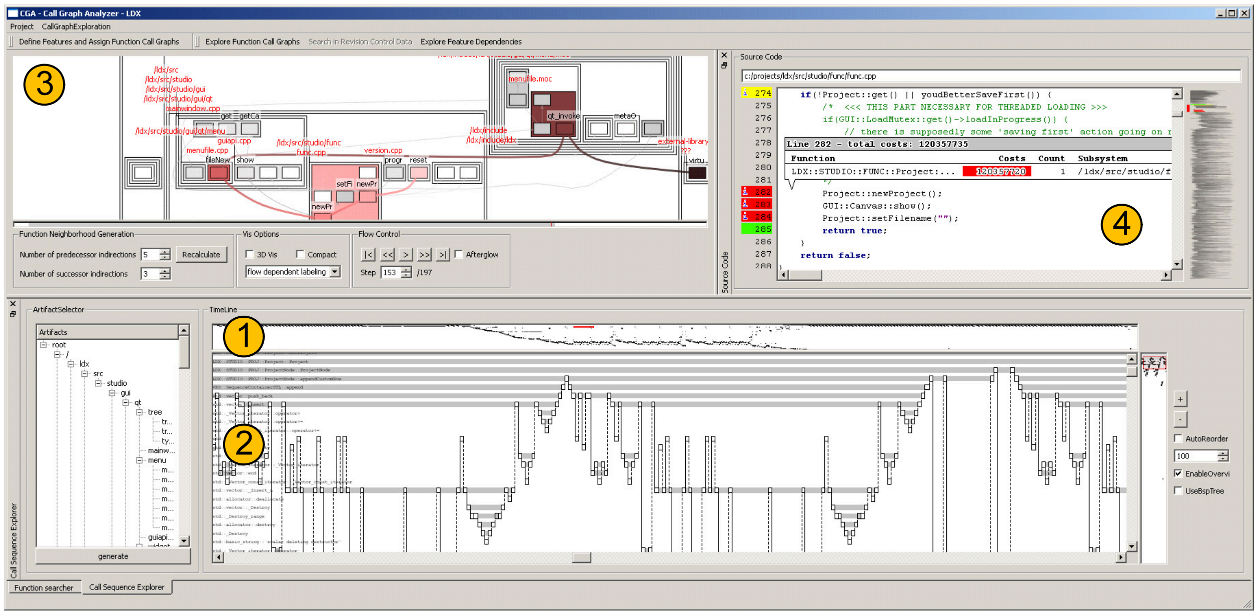
In this paper, we propose an analysis technique that combines information from dynamic execution traces with information on the hierarchical structuring of implementation units. Within integrated multiple views developers explore how control flow passes through system implementation during feature execution. Developers thereby receive support in understanding which parts of the implementation realize the feature's core functionality. The views focus on different aspects of execution traces: (1) A time-focused overview permits to detect different phases during execution. (2) A time-focused detail view allows understanding particular sequences of function interaction. (3) A collaboration-focused view permits to understand structural aspects of function interaction. Functions are thereby shown within their containment in the hierarchy of implementation units, which allows assessing the function's role in execution at higher-level abstraction. (4) A code-focused view enables developers to correlate findings made in graphical views with the corresponding code lines.

The key ideas of (a) using execution traces to reduce the amount of code a developer has to inspect for locating a feature and (b) showing the logged sequence of function calls within a high-level context are discussed extensively in recently published papers [e.g., 1]. The main contribution of this paper is to demonstrate the advantages of using an integrated visualization system that permits to explore feature execution within synchronized multiple views, each focusing on a different aspect of execution traces.

The analysis technique is implemented as a prototype tool that copes with large C/C++ production code (> MLOC).

[1] Throughout the paper, we use the term feature as a synonym for system functionality that can be triggered by the user and that produces an output visible to the user.

978-0-7695-3176-2/08 $25.00 © 2008 IEEE
DOI 10.1109/ICPC.2008.21


CGA - Call Graph Analyzer - LDX
Project
CallGraphExploration
Define Features and Assign Function Call Graphs
Explore Function Call Graphs
Search in Revision Control Data
Explore Feature Dependencies
Function Neighborhood Generation
Number of predecessor indirections
Number of successor indirections
Recalculate
Vis Options
3D Vis
Compact
Flow dependent labeling
Flow Control
Afterglow
Step
153
/197
Source Code
ArtifactSelector
Artifacts
generate
TimeLine
AutoReorder
EnableOvervi
UseBspTree
Function searcher
Call Sequence Explorer
1
2
3
4

c:/projects/ldx/src/studio/func/func.cpp

```
    if(!Project::get() || youdBetterSaveFirst()) {
        /*  <<< THIS PART NECESSARY FOR THREADED LOADING >>>
        if(GUI::LoadMutex::get()->loadInProgress()) {
            // there is supposedly some 'saving first' action going on r
        */
        Project::newProject();
        GUI::Canvas::show();
        Project::setFilename("");
        return true;
    }
    return false;
}
```

```
Line 282 - total costs: 120357735
Function                         Costs     Count  Subsystem
LDX::STUDIO::FUNC::Project:...   120357720     1  /ldx/src/studio/f
```

Figure 1: Screenshot of the analysis tool. Synchronizing (①, ②) time-, (③) collaboration-, and (④) code-focused views creates a rich user interface for exploring different characteristics of execution traces.

2. Fact Extraction

The tool's primary input data are sequences of function calls that are executed while the system runs the feature that shall be located within the code. For complex C/C++ software systems that either run on GNU/Linux operating system (OS) or on Microsoft Windows OS our tool provides means of extracting call sequences. Call chains of different threads are thereby separated. On both OS the extraction method is designed to be applied easily and to be non-intrusive. That is, neither source code instrumentation is necessary nor the system's build process needs to be altered significantly. This is an important usability requirement for an analysis tool that needs to be applied on a complex software system. For other OS, third party tools need to be consulted for call sequence extraction.

3. Views on Execution Traces

An execution trace is stored within our prototype analysis tool as integrated data structure that describes both the sequence of function calls and the containment of functions within a hierarchy of implementation units. When analyzing complex software systems, the amount of data stored in the data structures may be huge (number of functions > 10.000; number of sequence elements > 1.000.000). For effective exploration developers need to be given visualizations that provide views on reasonable cutouts of the complete data. The proposed analysis tool, therefore, implements various views that rely on different filtering techniques. The views are complementary in the sense that they focus on the trace's sequential (time-focused), structural (collaboration-focused), and code-related (code-focused) aspects (Figure 1).

Time-focused View

The analysis tool provides means of visualizing the time characteristics of execution traces explicitly and in a scalable way. The general layout of the view is inspired by UML sequence diagrams: Time is mapped to x-dimension. In y-dimension each row corresponds to a function. Function activity is encoded by coloring. At a coarse-grained level, explicit time visualization permits to detect various phases during feature execution. At a finer-grained level, visualizing time explicitly permits to understand in detail the sequence of function calls. The order of the function rows is thereby optimized for exploring the call stack in the given time window. The level of granularity can be adjusted seamlessly.

Collaboration-focused View

An important source of information that permits to assess a function's role within feature execution is the way it collaborates with other functions. Given a specific function in the focus of the developer's interest, the collaboration-focused view depicts a call graph consisting of all functions that are closely related to the specific function via calls. Context information on the functions, i.e., their containment within the

View	Suited for exploring…	Suited for selecting and querying details on …
Time-focused Overview	… execution phases … time ranges where code statements of selected functions are executed … time ranges where code statements of selected implementation units are executed	… points in time … time ranges
Time-focused Detail View	… sequences of calls … active functions in time range … call stack in time range … function interaction patterns	… functions … calls … points in time … time ranges
Collaboration-focused View	… call relations between functions … how control flow passes through higher-level implementation units … call stack at given point in time	… functions … higher-level implementation units … calls
Code-focused View	… executed statements … how call statements are resolved at runtime	… calls

Table 1: Summary of the views' main exploration purposes and the kinds of artifacts that can be selected within them.

hierarchy of implementation units, permits to assess the functions' purposes at higher-level abstraction. Additionally, when having selected a point in time, the corresponding call stack is visually superimposed onto the call graph.

Code-focused View

The goal of the code focused view is to enable developers to trace findings obtained within graphical views down to code locations. Similar to slice viewers, code statements are superimposed by trace information such as execution costs – thereby, implicitly showing code coverage. Additionally, call statements are annotated with the names of the dynamically resolved functions, thereby, revealing concrete calls covered by polymorphism or function pointers.

Synchronizing Views

Synchronizing time-, collaboration-, and code-focused views creates a rich user interface for both exploring specific aspects of execution traces and selecting artifacts for querying further details. Table 1 summarizes (a) the main exploration purposes of each view and (b) artifacts that can be selected within the views to query details on them. Due to synchronization, query results are shown in all suitable views. Linkage between time-focused and collaboration-focused views is particularly useful in two ways:

- By moving the currently selected "time cursor" within a time-focused view, developers experience in the collaboration-focused view how control flow passes through functions and higher-level implementation units.
- Selecting functions or higher-level implementation units within the collaboration-focused view highlights in the time-focused views when code of these units has been executed.

The latter, understanding when specific functions or higher-level implementation units are active within the full trace, permits to identify reasonable time ranges to analyze in detail next.

4. Related Work

Various concepts exist to visually support developers in locating features in unfamiliar code. Most of them present results from fact analysis as "timeless" aggregated high-level graphs showing collaboration between code artifacts at various levels of abstraction [2, 3, 4, 6, 8, 9, 12, 13]. Some of these concepts [2, 6, 8, 9], as well as further ones [5, 7, 10] additionally depict time explicitly. Our concept differs from the proposed collaboration-focused visualization techniques in that our views center on specific functions and their interactions within local scope, however, additionally providing a global (implementation unit) context. In fact, this approach is largely complementary to an approach of visualizing high-level graphs. An analysis tool should reasonably provide both kinds of visualizations. With respect to existing time-focused visualizations, our concept differs from the mentioned ones in that it provides means of seamless navigation from viewing the execution trace in full down to viewing detailed function interaction. Cornelissen et al. [2] describe how to use time-focused views as user interface for obtaining reduced collaboration-focused views. We extend this approach: By extensive synchronization each view serves as a user interface for querying details on trace artifacts and results are shown in all other views. Even the code view, for instance, showing just a small fraction of the trace data, i.e., one function and outgoing calls, can be used as user interface to analyze all occurrences of a call within the full trace.

5. Case Study

With our partner *3D Geo GmbH* we recently analyzed their *LandXplorer Studio* software system for management and visualization of geovirtual 3D city models. The C/C++ system is complex in the sense that it consists of approximately 700.000 lines-of-code and that it is subject to development and enhancements for over 10 years. Our analysis tool was applied to facilitate enhancing LandXplorer's feature of performing distance queries within geovirtual models. Distance measures were given in meters only and should be additionally shown in miles on demand. Hence, the task was to locate the code responsible for setting the distance labels' texts and then introduce a code change that permits to switch between different metrics. To give an impression of LandXplorer's code complexity, we briefly report results here obtained via static pattern matching: The pattern *distance* is found 892 times in 152 different files (*.h, *.cpp, *.c) – *label* is found 1891 times in 250 different files.

Applying the extraction mechanism of our tool while running a distance query, resulted in an execution trace consisting of ~1500 different functions and ~300.000 call entries and exits. With the exploration facilities of the tool, we were able to locate the code locations defining the distance labels' texts in approximately 30 minutes with having had only a very rough understanding of the system's coding before. After having found these code locations, we could start extending the system to support multiple distance measures.

6. Conclusions and Future Work

Exploring execution traces seems to be an important step to facilitate feature location. However, execution traces are typically large and exploring them represents a problem of its own. Providing multiple synchronized views on a trace seems to be of great help for developers during the exploration task, since synchronization enables them to perform a loop-like exploration from overview to detailed view and back to overview. That is, they start from choosing a time range in the time-focused overview. Then, the time range is analyzed within more detailed views. Selecting an artifact in one of those detailed views, permits to understand within the overview when the artifact is active throughout the full trace. This global perspective, however, enables developers to choose further time ranges that should reasonably be analyzed in detail next. As future work, we will improve the tool by incorporating analysis techniques such as trace comparison [14]. By this, additional hints are provided that help developers to prioritize which time range to analyze in detail first.

7. References

[1] J. Bohnet and J. Döllner. Visual exploration of function call graphs for feature location in complex software systems. *ACM Symp. on Software Visualization*, 2006, pp. 95-104.

[2] B. Cornelissen, D. Holten, A. Zaidman, L. Moonen, J. J. van Wijk, and A. van Deursen. Understanding Execution Traces Using Massive Sequence and Circular Bundle Views. *Int'l Conf. on Program Comprehension*, 2007, pp. 49-58.

[3] J. Gargiulo and S. Mancoridis. Gadget: A Tool for Extracting the Dynamic Structure of Java Programs. *Int'l Conf. on Software Engineering & Knowledge Engineering*, 2001, pp. 244-251.

[4] T. Jacobs and B. Musial. Interactive visual debugging with UML. *ACM Symp. on Software Visualization*, 2003, pp. 115-122.

[5] D. Jerding and S. Rugaber. Using Visualization for Architectural Localization and Extraction. *Working Conf. on Reverse Engineering*, 1997, pp. 56-65.

[6] D. B. Lange and Y. Nakamura. Object-Oriented Program Tracing and Visualization. *IEEE Computer*, 30(5), 1997, pp. 63-70.

[7] K. Lukoit, N. Wilde, S. Stowell, and T. Hennessey. TraceGraph: Immediate Visual Location of Software Features. *Int'l Conf. on Software Maintenance*, 2000, pp. 33-39.

[8] B. Malloy and J. F. Power. Exploiting UML dynamic object modeling for the visualization of C++ programs. *ACM Symp. on Software Visualization*, 2005, pp. 105-114.

[9] A. Marburger and B. Westfechtel. Tools for understanding the behavior of telecommunication systems. *Int'l Conf. on Software Engineering*, 2003, pp. 430-441.

[10] L. Martin, A. Giesl, and J. Martin. Dynamic Component Program Visualization. *Working Conf. on Reverse Engineering*, 2002, pp. 289-298.

[11] A. von Mayrhauser and A. M. Vans. Program Comprehension During Software Maintenance and Evolution. *IEEE Computer*, 28(8), 1995, pp. 44-55.

[12] M. Salah, S. Mancoridis, G. Antoniol, and M. Di Penta. Scenario-Driven Dynamic Analysis for Comprehending Large Software Systems. *Int'l Conf. on Software Maintenance and Reengineering*, 2006, pp. 71-80.

[13] M. P. Smith and M. Munro. Runtime Visualisation of Object Oriented Software. *Int'l Workshop on Visualizing Software for Understanding and Analysis*, 2002, pp. 81-89.

[14] N. Wilde and M. C. Scully, Software reconnaissance: mapping program features to code. *Journal of Software Maintenance: Research and Practice*, 7(1), 1995, pp. 49-62.